Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА ДВОВИМІРНОЇ ГРИ «ЛЕГЕНДА ПРО КАЇНА» З ВИКОРИСТАННЯМ UNITY

DEVELOPMENT OF THE 2D GAME "LEGEND OF CAIN" USING UNITY

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-42
Рева Нікіта Дмитрович

Керівник:
Гордєєв Олександр Олександрович

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«20» 12 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Реві Нікіті Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка двовимірної гри «Легенда про Каїна» з використанням Unity».*
Керівник роботи: *д.т.н., професор Гордєєв О. О.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р.* № *474/0-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *Unity, Visual Studio, методичні вказівки до виконання кваліфікаційної роботи бакалавра.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз предметної області, постановка завдання, визначення вимог, реалізація інтерфейсу, логіка, тестування та виправлення помилок.*

5. Перелік графічного матеріалу: *8 рисунків*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Гордєєв О. О.*		
Специфікація вимог до розробленої системи	*Гордєєв О. О.*		
Розробка об'єкта проектування	*Гордєєв О. О.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	0,57 %		
Академічна доброчесність	*Гордєєв О. О.*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	вик.
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 01.03.2025 р.	вик.
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	вик.
4	Розробка функціонально-структурної схеми роботи об'єкта проектування	до 29.03.2025 р.	вик.
5	Практична реалізація об'єкта проектування	до 26.04.2025 р.	вик.
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	вик.
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	вик.

Здобувач вищої освіти _____ Нікіта РЕВА

Керівник кваліфікаційної роботи _____ Олександр ГОРДЄЄВ

АНОТАЦІЯ

Рева Н. Д. Розробка двовимірної гри «Легенда про Каїна» з використанням Unity. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел.

У першому розділі здійснено аналіз предметної області, визначено актуальність теми та сформульовано завдання кваліфікаційної роботи. У другому розділі розглянуто вимоги до програмного забезпечення та спроектовано архітектуру майбутньої гри. У третьому розділі описано процес реалізації ігрового застосунку: розробку механік, інтерфейсу, навігації, анімації та тестування. У висновках наведено результати роботи, оцінено ефективність реалізації та перспективи подальшого розвитку проекту.

Ключові слова: Unity, C#, Visual Studio, спрайти, скрипти, 2D-гра, ігровий рушій, програмна архітектура, геймдизайн, анімації.

ABSTRACT

Reva N. Development of the 2D Game "Legend of Cain" Using Unity. Manuscript. Bachelor's qualification thesis of the educational program "Software Engineering" specialty "Software Engineering" Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions, a list of references, and appendices.

The first chapter presents an analysis of the subject area, outlines the relevance of the topic, and formulates the objectives of the thesis. The second chapter discusses the software requirements and presents the architecture design of the future game. The third chapter describes the implementation process of the game application, including the development of gameplay mechanics, user interface, navigation, animation, and testing. The conclusions summarize the results of the work, evaluate the implementation effectiveness, and outline the prospects for further development of the project.

Keywords: Unity, C#, Visual Studio, sprites, scripts, 2D game, game engine, software architecture, game design, animations.

ЗМІСТ

ВСТУП

Актуальність теми. Ігрова індустрія продовжує активно розвиватися, охоплюючи дедалі більше користувачів та створюючи попит на високоякісні проєкти, що поєднують ігрову механіку, художнє оформлення та технічну реалізацію. Особливою популярністю користуються двовимірні рольові ігри (2D RPG), які дають змогу гравцям зануритись у вигаданий світ із розгалуженою структурою рівнів, ворогами, прокачуванням персонажа та дослідженням середовища. Розробка подібної гри вимагає комплексного підходу до програмування, дизайну, сценарної побудови та організації сцени.

Метою даної кваліфікаційної роботи є створення повноцінної 2D-рольової гри «Легенда про Каїна» із використанням Unity та мови програмування C#, яка включає реалізацію процедурної генерації рівнів, бойової системи, елементів інтерфейсу та логіки переходу між ігровими сценами.

Об’єктом кваліфікаційної роботи є процес дослідження та створення програмного забезпечення для інтерактивної 2D-гри.

Предметом кваліфікаційної роботи є архітектура, логіка та методи реалізації функціоналу гри в середовищі Unity.

Для досягнення поставленої мети були визначені такі завдання:

- проаналізувати наявні рішення у сфері 2D-геймдеву;
- сформувати технічне завдання та вимоги до функціоналу;
- обрати технологічні засоби для розробки;
- реалізувати управління персонажем;
- створити бойову систему;
- розробити базовий штучний інтелект ворогів;
- спроєктувати ігрові локації;
- відрегулювати систему сцен;
- провести тестування застосунку та виправити виявлені помилки.

РОЗДІЛ 1
АНАЛІЗ СУЧАСНИХ РІШЕНЬ У СФЕРІ РОЗРОБКИ ІГОР ТА ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Рольові ігри у двовимірному просторі зберігають стабільну популярність серед розробників та гравців завдяки своїй відносній простоті реалізації, широким можливостям для творчості та глибокому зануренню у сюжет. Основною особливістю таких ігор є взаємодія гравця з віртуальним світом, що включає дослідження територій, ведення боїв із супротивниками, розвиток персонажа, виконання завдань та збір ресурсів.

За останні роки відеоігрова індустрія значно розширила свої технологічні межі. Сучасні 3D-проєкти вимагають надскладної підготовки: створення високополігональних моделей персонажів та оточення, налаштування складного освітлення, розробки фізично коректних систем зіткнень та оптимізації рендерингу. Для підтримки таких процесів необхідне потужне обладнання та широкий штат фахівців, що ускладнює реалізацію навіть середніх за масштабом проєктів малими студіями й інді-розробникам.

2D-ігри, у свою чергу, є більш доступними для реалізації та постійно породжують геніїв-ентузіастів, які зосереджуються на ігровій механіці, сюжеті та художньому стилі, вкладають душу в свою розробку та прислухаються до ком’юніті. Ігри на кшталт «Katana Zero», «Terraria», «Cuphead», «Enter the Gungeon», «Ori and the Will of the Wisps», доводять, що 2D-геймплей може бути не менш глибоким, ніж у 3D-проектах, а зараз, коли ААА ігри стають (багато мільйонні розробки) стають об’єктом розчарування та насмішок геймерів, через свою криву реалізацію, застарілі ідеї, тощо, 2D проекти стабільно радують поціновувачів і все більше конкурують з титанами індустрії за своє місце в топах.

Unity універсальний ігровий рушій, що дозволяє створювати, ігрові застосунки, як повноцінним студіям, так і звичайним, школярам. Його

компоненти, що вбудовані всередину і доступні безкоштовно, дозволяють створювати освітлення, колізію, анімації для гри, також Unity підтримує потужну і мову програмування C#, яка не лише є ООП мовою, а ще й дозволяє безпечно працювати з пам’яттю і є дуже гнучкою.

Зазвичай основні проблеми при розробці 2D ігор, це знайти свій сетінг та спосіб реалізувати ті чи інші задачі, інколи від кількості доступних рішень розбігаються очі.

У межах кваліфікаційної роботи особливий акцент зроблено на вивченні технологічних рішень, які сприяють ефективному створенню двовимірної гри з чіткою структурою, продуманим управлінням ігровими станами та інтуїтивною взаємодією з користувачем. Саме ці аспекти – архітектура, геймплей і UX – формують ядро сучасної 2D-розробки, де художній стиль і механіка є не менш важливими за технічну реалізацію.

Ринок відеоігор наразі переживає черговий етап активного розвитку, при цьому 2D-сегмент зберігає стійку популярність. Причиною цього є кілька чинників: з одного боку – простота технічної реалізації порівняно з 3D-проєктами, а з іншого – доступність інструментів, які дозволяють сконцентруватися на оригінальності задуму, геймдизайні та художньому оформленні. Сучасні 2D-ігри вже давно перестали асоціюватися виключно з ретро-естетикою – сьогодні це повноцінні продукти з глибоким сюжетом, динамічними механіками та привабливою графікою.

Прикладом слугують такі хітові проєкти, як «Katana Zero», «Cuphead», «Enter the Gungeon», «Ori and the Will of the Wisps». Вони демонструють, що інноваційний підхід до 2D-геймплею, зосереджений на атмосфері, дизайні рівнів і складності, здатен принести не лише схвальні відгуки критиків, але й комерційний успіх. Це надихає інді-розробників на створення власних, авторських ігор, де саме творча складова виходить на перший план, зазвичай такі ігри навіть надихають великі ігрові студії на додавання якихось нових механік.

Для реалізації таких проєктів надзвичайно важливим є вибір рушія.

Unity – один із найпотужніших і водночас доступних інструментів, який дозволяє створювати 2D-ігри на професійному рівні, при цьому він є безкоштовним та легким в освоєнні. Його переваги включають:

– підтримку двовимірної фізики, анімацій, партиклів і тайлмапів «із коробки»;

– компонентну модель побудови ігрових об’єктів, що дозволяє структурувати логіку, забезпечуючи високу гнучкість та повторне використання коду;

– повну інтеграцію з мовою C#, яка є зручною та потужною для реалізації логіки гри, з можливістю використання шаблонів проєктування та зовнішніх бібліотек;

– підтримку багатоплатформенності (Windows, Android, iOS, WebGL, тощо), що забезпечує широкий спектр поширення готового продукту.

Unity також надає змогу реалізовувати складні системи управління станами – наприклад, через стейт-машини, які можуть використовуватись для реалізації поведінки ворогів, перемикань між анімаціями, логіки меню тощо. Завдяки інструментам на кшталт Animator, ScriptableObject і UnityEvent можна вибудовувати масштабовану та легко підтримувану архітектуру проєкту.

Ще однією перевагою є можливість створення UI будь-якої складності за допомогою Canvas-системи з анімованими переходами, адаптивною верстkою під різні роздільні здатності та інтерактивними елементами, які дозволяють створити зручний та інтуїтивно зрозумілий інтерфейс.

Значну роль у сучасному підході до розробки відіграє також система контролю версій (наприклад, Git), яка дозволяє працювати в команді, відстежувати зміни та зберігати історію проєкту. У поєднанні з автоматизованим тестуванням, CI/CD-практиками та профайлінгом продуктивності це формує повноцінне середовище для створення якісного програмного продукту.

У підсумку, сучасна розробка 2D-ігор є складним і багатогранним процесом, у якому переплітаються творчість, інженерна думка та користувацький досвід. Інструменти на кшталт Unity суттєво спрощують

технічну сторону процесу, дозволяючи зосередитися на головному – створенні гри, яка буде цікавою та захопливою для гравця.

Індустрія 2D-ігор залишається актуальною навіть у період стрімкого розвитку тривимірної графіки, віртуальної та доповненої реальності. Простота реалізації, менші апаратні вимоги, висока виразність візуального стилю та можливість швидкого створення прототипів сприяють поширенню двовимірних проєктів, особливо серед інді-розробників.

На цифрових платформах, таких як Steam, Itch.io, Google Play та App Store, щоденно з’являються десятки нових 2D-ігор, значну частину яких створено невеликими командами або навіть однією особою. Це стало можливим завдяки доступності сучасних рушіїв, наявності відкритих ресурсів та демократизації інструментів розробки.

Розробка навіть найпростіших 2D-проєктів потребує знань у низці напрямів: геймдизайну, програмування, візуальної композиції, анімації, користувацького досвіду, роботи з ресурсами, звуком та тестування.

Серед найпопулярніших інструментів для реалізації двовимірних ігор є рушій Unity, який надає потужне та зручне середовище для розробки:

– підтримка 2D-фізики, спрайтів, анімації, Tilemap, UI, звуку, частинок;

– наявність вбудованого редактора сцен та інтерфейсів з адаптивною верстkою;

– можливість кросплатформеного експорту з одного проєкту;

– використання мови C# з об’єктно-орієнтованим підходом і підтримкою шаблонів проєктування;

– підтримка архітектури ECS (Entity-Component-System), що спрощує масштабування гри;

– наявність системи навігації (NavMesh) навіть для 2D-проєктів;

– візуальні редактори для анімації та взаємодії об’єктів (Animator, Timeline, Cinemachine).

Unity також має безкоштовну ліцензію для розробників із невеликим доходом, що відкриває можливості для студентів, ентузіастів і невеликих команд

без фінансових витрат. Це дозволяє зосередитися на творчій реалізації ідей та практичному вдосконаленні навичок.

Ще однією перевагою 2D-ігор є їх доступність для користувачів: такі ігри не потребують високої обчислювальної потужності та можуть запускатися навіть на старих ПК або бюджетних мобільних пристроях. Завдяки цьому 2D-проєкти мають значно ширшу потенційну аудиторію, особливо в умовах глобального ринку.

Багато класичних жанрів – платформери, головоломки, рогалики, аркади – традиційно реалізуються у двовимірному просторі, що надає змогу експериментувати з механіками без складної тривимірної логіки. До того ж, художній стиль у 2D-графіці може бути не менш виразним, ніж у 3D, а іноді й більш емоційним та атмосферним завдяки піксель-арту або стилізованій графіці. Проте розробка 2D-гри має і свої виклики:

– забезпечення стабільної роботи на різних пристроях та платформах;

– правильна побудова ігрової архітектури, що дозволяє масштабування без критичного зростання складності;

– оптимізація використання ресурсів: текстур, анімацій, аудіо;

– розробка UI, який адаптується під різні роздільності;

– підтримка збереження прогресу та системи досягнень.

Отже, створення повноцінної 2D-гри є водночас викликом і чудовою нагодою для демонстрації практичних знань у сфері програмного забезпечення. Розробка власного 2D-проєкту є актуальним напрямом дослідження в межах кваліфікаційної роботи бакалавра, який поєднує інженерні навички з креативністю.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

У межах дослідження розглядалося питання створення комп’ютерної гри у двовимірному просторі, що поєднує ключові риси пригодницького та рольового жанрів. Основним інструментом розробки обрано ігровий рушій

Unity, а як мова програмування – C#. Проєкт мав на меті реалізацію інтерактивного середовища з базовими механіками дослідження, бою, взаємодії з об’єктами сцени та керуванням переходами між ігровими станами. Для досягнення поставленої мети було визначено такі основні завдання:

– провести аналітичний огляд сучасних підходів до розробки 2D-ігор, зокрема механік у жанрі RPG; дослідити типові структури ігрових сцен, поведінку неігрових персонажів, організацію анімацій та логіку гравця;

– визначити перелік функціональних і нефункціональних вимог до гри, серед яких: продуктивність, адаптивність інтерфейсу, масштабованість, гнучкість архітектури, стабільність логіки управління та зручність взаємодії для кінцевого користувача;

– обґрунтувати вибір технологій та засобів розробки: рушія для побудови рівнів і реалізації анімацій, фізики та UI; мови програмування; середовища для написання коду; додаткових інструментів – систем тайлмапів, анімаційних контролерів, навігаційної сітки та інтерфейсного полотна;

– реалізувати механіку управління ігровим персонажем з підтримкою пересування, взаємодії з об’єктами, реакції на зіткнення, орієнтації у просторі сцени та активації тригерів;

– побудувати базову бойову систему, яка включає завдання шкоди, отримання ушкоджень, зміну станів персонажа в бою, а також обробку смерті;

– створити логіку поведінки супротивників, що включає перехід між різними станами: спокій, патрулювання, переслідування, атака; реалізувати відповідні анімації;

– розробити ігрові локації з використанням кількох типів тайлів, включаючи покриття поверхні, перепони, водойми та декоративні елементи; організувати кордони рівня та обмеження руху;

– налаштувати систему сцен: початкове меню, ігрову сцену, екран паузи, перехід між рівнями та завершення гри; реалізувати передачу даних між сценами та плавне їх завантаження;

– повноцінне тестування всіх реалізованих механік, включно з боєм,

анімацією, інтерфейсом, переходами між станами, перевіркою на помилки та продуктивністю.

У процесі реалізації використовувались профільні навчальні ресурси, технічна документація рушія, а також приклади з відкритих джерел, що демонструють підходи до побудови гнучкої ігрової логіки.

Окрему увагу було приділено внутрішній архітектурі програмного коду: структура проєкту організована за принципами модульності, повторного використання, масштабованості та розділення відповідальності. Компоненти гри побудовані з урахуванням можливості подальшого розширення функціоналу, оптимізації продуктивності та адаптації під різні платформи.

У результаті виконання поставлених завдань створено базову версію інтерактивної 2D-гри, що демонструє реалізацію типових механік пригодницького жанру та може слугувати основою для подальшого розвитку. Такий підхід дозволяє не лише перевірити застосування теоретичних знань на практиці, але й закласти основу для майбутніх проєктів у галузі інженерії програмного забезпечення, орієнтованої на розробку ігор.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

На початковому етапі розробки програмного продукту важливою задачею стало визначення загальної концепції, функціональних можливостей і технічних обмежень майбутньої гри. З урахуванням жанрової належності до двовимірних рольових ігор з елементами бою та дослідження середовища, особлива увага була зосереджена на формуванні чітких функціональних і нефункціональних вимог до системи.

Для обґрунтування архітектурних рішень було проаналізовано низку відомих ігор, створених на сучасних рушіях, зокрема Death Must Die, Enter the Gungeon і Hades. Кожен із цих проєктів відзначається високим рівнем реалізації геймплейних механік, плавною анімацією та чіткою логікою станів об'єктів. Їх спільними характеристиками є:

– модульна побудова ігрової логіки, що дозволяє легко розширювати функціональність без переробки вже реалізованих систем;

– акцент на візуальну виразність за допомогою ручної або стилізованої анімації;

– підтримка складної взаємодії з оточенням і широкого спектра предметів або ворогів;

– інтуїтивний інтерфейс, орієнтований на швидке сприйняття гравцем поточних дій і стану персонажа.

З урахуванням зазначених прикладів, у межах дослідження було сформульовано функціональні вимоги, відповідно до яких ігровий продукт має забезпечити:

– можливість вільного пересування по ігрових рівнях;

– інтеракцію з об’єктами сцени та неігровими персонажами;

– реалізацію бойової системи з відгуком на дії користувача, шкодою, станами персонажа та ефектами;

– підтримку переходу між різними локаціями зі збереженням цілісності ігрового процесу.

Окремо були визначені нефункціональні вимоги, зокрема:

– стабільність продуктивності гри на типових користувацьких пристроях;

– розширюваність архітектури без необхідності внесення глибоких змін до існуючих модулів;

– логічна організація коду, що відповідає принципам підтримуваності, модульності та повторного використання;

– швидкий час відгуку інтерфейсу, плавність переходів і зручність у взаємодії з елементами керування.

Загальне структурування вимог на початковому етапі дало змогу сформувати цілісну та логічно організовану концептуальну модель проєкту ще до переходу до безпосередньої реалізації. Саме цей підхід дозволив не лише визначити перелік основних функціональних можливостей гри, а й детально описати взаємозв’язки між майбутніми компонентами, передбачити шляхи їхнього розвитку та розширення. Ретельно опрацьовані вимоги стали своєрідною дорожньою картою для кожного етапу розробки: від планування архітектури до створення окремих класів, систем і модулів. Водночас, це дозволило завчасно виявити потенційно складні місця, які могли б призвести до архітектурних помилок чи труднощів із масштабуванням, і знайти оптимальні рішення ще до того, як ці труднощі стали б актуальними.

У результаті було побудовано комплексну архітектуру гри, яка передбачає чіткий розподіл відповідальності між окремими класами, підсистемами та модулями. Кожен клас виконує вузькоспеціалізовану функцію і взаємодіє з іншими елементами системи через чітко визначені інтерфейси. Така структура забезпечує як простоту підтримки, так і легкість у подальшій модифікації чи масштабуванні гри. Кожна підсистема – управління гравцем, бойова система, AI ворогів, логіка переходу між сценами, інтерфейс

користувача, обробка анімацій – ізольована від інших настільки, наскільки це можливо, що мінімізує ризик виникнення побічних ефектів при зміні окремих елементів.

Структурна взаємодія базових елементів системи, а також їхня ієрархія, підпорядкованість та логічна організація наведені на UML-діаграмі (рис. 2.1).

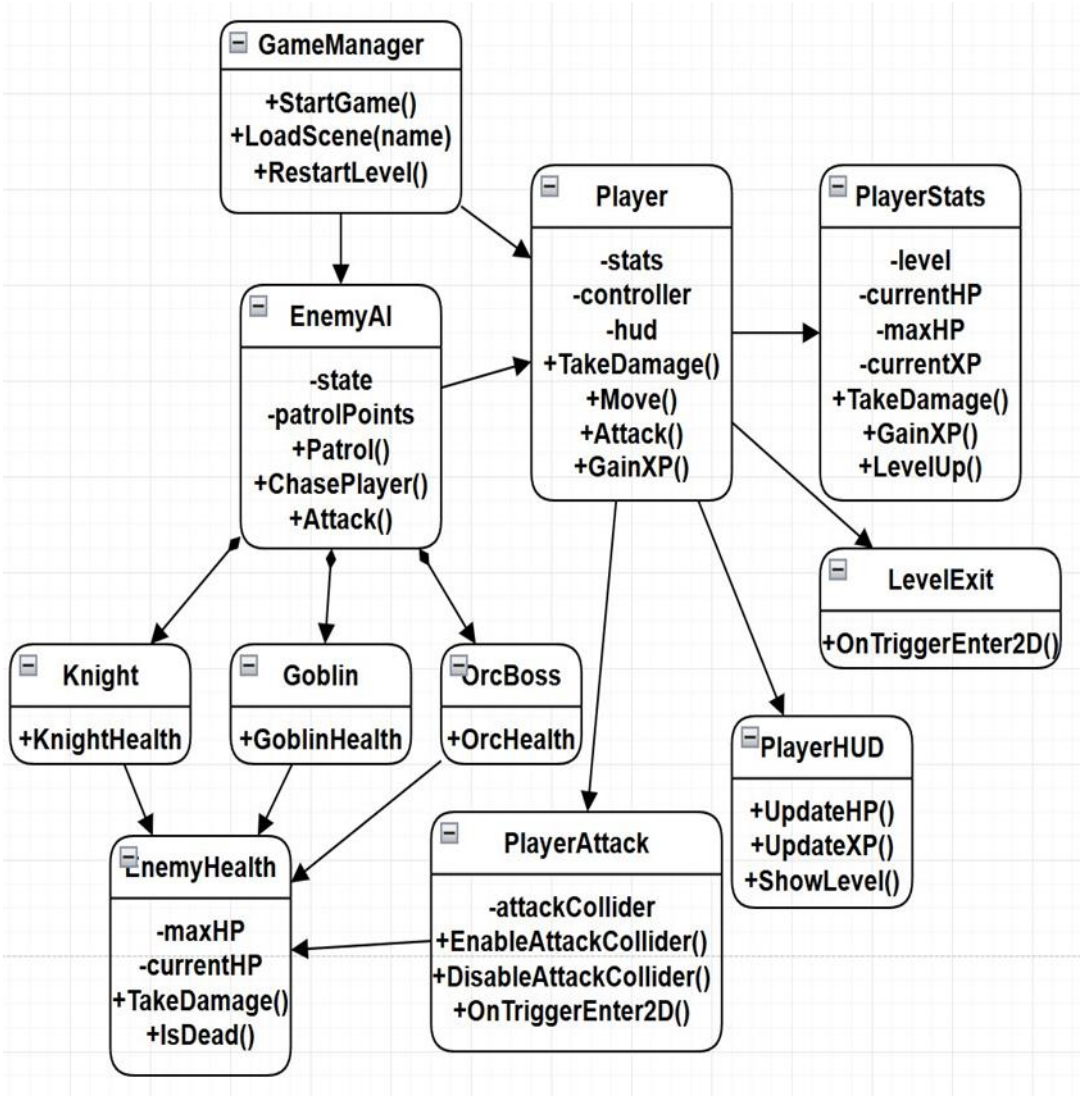

Рисунок 2.1 – Компонентна структура гри

Саме ця діаграма є візуалізацією розробленої архітектури і дозволяє наочно побачити, як між собою пов’язані всі основні компоненти гри, з яких класів складається кожен модуль та яким чином реалізовано обмін даними між окремими частинами проєкту. Такий підхід є не лише ознакою якісної

підготовки до розробки, а й гарантією стабільності, надійності й масштабованості створеної гри в майбутньому. Під час розробки механіки гри окрему увагу було приділено зручності розширення. Наприклад, система ворогів реалізована так, що дозволяє легко додавати нові типи противників із різною поведінкою без зміни основного коду гри.

При розробці архітектури гри я застосував модульний підхід, при якому основні елементи логіки винесені в окремі скрипти. Зокрема:

– PlayerController – обробка керування рухом, анімаціями ходьби та атаки гравця (рис. 2.2);

Рисунок 2.2 – Зовнішній вигляд персонажа

– EnemyAI – логіка руху, виявлення гравця, патрулювання (рис. 2.3);

Рисунок 2.3 – Зовнішній вигляд базового ворога

– PlayerStats – зберігає HP, XP, Level, реалізує шкоду та прокачку;
– PlayerAttack – керує зоною атаки, обробляє нанесення шкоди ворогу;
– EnemyHealth – зберігає HP ворога, обробляє отримання шкоди;

– LevelExi – реалізація переходів між сценами;

– LevelGenerator – генерує ігрову карту.

У процесі визначення вимог до гри «Легенда про Каїна» важливу роль відігравало проектування логічної структури сцен. Кожна ігрова локація реалізується як окрема сцена Unity, яка динамічно завантажується при вході гравця у відповідну зону переходу. Це дозволяє уникнути надмірного навантаження на одну сцену зайвими об’єктами, оптимізувати використання пам’яті й забезпечити стабільну продуктивність навіть у присутності кількох ворогів, анімацій та елементів інтерфейсу користувача.

Архітектура проєкту спроєктована так, щоб забезпечити гнучке розширення функціоналу. Всі основні системи реалізовані як незалежні модулі: наприклад, інтерфейс користувача (HUD) не залежить від логіки руху персонажа, а обробка зіткнень не пов’язана безпосередньо із системою діалогів. Це дає змогу додавати нові типи ворогів чи ігрові механіки без втручання в базовий код та дозволяє працювати над окремими компонентами ізольовано, зберігаючи цілісність і підтримуваність проєкту.

Ініціалізація головного героя виконується під час запуску відповідної сцени. На цьому етапі скрипт гравця зберігає посилання на основні компоненти: фізику (Rigidbody2D), систему анімацій (Animator) та спрайтову візуалізацію (SpriteRenderer). Така структура дозволяє гравцю коректно взаємодіяти з оточенням, відтворювати анімації руху та зміни станів, а також оперативно реагувати на події у грі (ліст. 2.1).

Лістинг 2.1 – Ініціалізація головного героя

```
private void Awake()
{
    Instance = this;
    rb = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
    spriteRenderer = GetComponent<SpriteRenderer>();
}
```

кінець лістингу 2.1

Реалізація переміщення головного героя виконується за допомогою методу Rigidbody2D.MovePosition, що забезпечує плавне та коректне пересування персонажа згідно з напрямком, який визначається гравцем. Введення обробляється у методі Update, а фізичне переміщення здійснюється у FixedUpdate для уникнення ривків та забезпечення стабільної взаємодії з фізичними колайдерами середовища. Додатково, стан бігу або зупинки впливає на анімацію персонажа, що забезпечує візуальний зворотний зв'язок гравцеві. Завдяки такій структурі руху, герой реагує на натискання клавіш у реальному часі, а сам процес переміщення виглядає природно навіть у динамічних ситуаціях, наприклад, під час бою або уникнення ворогів. Це створює відчуття керованості та підсилює загальну ігрову динаміку, роблячи геймплей комфортним та передбачуваним для користувача (ліст. 2.2).

Лістинг 2.2 – Рух головного героя

```
private void Update()
{
    if (isDead) return;
    moveInput = GameInput.Instance.GetMovementVectorNormalized();
    if (moveInput.x < 0)
        spriteRenderer.flipX = true;
    else if (moveInput.x > 0)
        spriteRenderer.flipX = false;
    animator.SetBool("isRunning", moveInput != Vector2.zero);
}
private void FixedUpdate()
{
    if (!isDead && !isHurt && moveInput != Vector2.zero)
        rb.MovePosition(rb.position + moveInput * (moveSpeed *
Time.fixedDeltaTime));
}
```

кінець лістингу 2.2

Механізм отримання шкоди реалізовано через зменшення значення поточного здоров'я. Коли персонаж отримує удар, його health зменшується на задану величину, запускається анімація пошкодження, а також вмикається тимчасова невразливість – протягом цього часу гравець не може отримати шкоду повторно. Такий підхід дозволяє уникнути моментальної загибелі від

серії ударів і дає час для ухилення або зміни тактики. Додатково, у разі зниження здоров’я до нуля викликається логіка смерті (ліст. 2.3).

Лістинг 2.3 – Отримання шкоди гравцем

```
public void TakeDamage(int damage)
{
    if (isDead) return;
    currentHP -= damage;
    if (currentHP > 0)
    {
        if (animator) animator.SetTrigger("HurtTrig");
    }
    else
    {
        Die();
    }
}
```

кінець лістингу 2.3

Механізм підвищення рівня реалізовано у вигляді окремого методу LevelUp(). Після досягнення необхідної кількості досвіду у гравця автоматично збільшується рівень, максимальний запас здоров’я та сила атаки. Досвід обнуляється, а здоров’я повністю відновлюється. Також можна вручну через інструменти Unity змінювати показники на ті, які тобі будуть потрібні в конкретній ситуації. Такий підхід дозволяє підтримувати мотивацію гравця до подальшого розвитку й забезпечує поступове ускладнення ігрового процесу (ліст. 2.4).

Лістинг 2.4 – Підвищення рівня гравця

```
private void LevelUp()
{
    level++;
    maxHealth += 10;
    damage += 2;
    experience = 0;
    currentHealth = maxHealth;
}
```

кінець лістингу 2.4

Зміна напрямку погляду головного героя у грі реалізована шляхом відстеження напрямку руху персонажа. Якщо гравець натискає клавішу руху вліво, спрайт автоматично віддзеркалюється по горизонталі, і герой повертається обличчям ліворуч. Якщо рух відбувається вправо, спрайт повертається в початковий стан. Це потребувало допрацювання розташування колізії зброї гравця. Такий підхід забезпечує інтуїтивну відповідність між керуванням і візуальним зображенням персонажа, а також коректне відображення анімацій і дозволяє гравцю краще відчувати те що відбувається в грі (ліст. 2.5).

Лістинг 2.5 – Зміна напрямку погляду гравця

```
private void Update()
{
    moveInput = GameInput.Instance.GetMovementVectorNormalized();
    if (moveInput.x < 0)
        spriteRenderer.flipX = true;
    else if (moveInput.x > 0)
        spriteRenderer.flipX = false;
}
```

кінець лістингу 2.5

Смерть головного героя реалізована у компоненті Health. Коли здоров’я персонажа (currentHP) стає меншим або дорівнює нулю, викликається метод Die(). Він встановлює прапорець isDead, запускає анімацію смерті через тригер DeathTrig та знищує об’єкт через одну секунду, даючи час для завершення анімації. Повторна обробка смерті виключена – метод виходить, якщо герой уже мертвий (ліст. 2.6).

Лістинг 2.6 – Реалізація смерті головного героя

```
private void Die()
{
    if (isDead) return;
    isDead = true;
    if (animator != null)
    {
        animator.SetTrigger("DeathTrig");
        Destroy(gameObject, 1f);
```

```
    }
    else{
        Destroy(gameObject);
        }
}
```

кінець лістингу 2.6

Атака ворога реалізована через активацію колайдера зброї або вбудованої зони ураження під час наближення до гравця. Логіка побудована так, що після перевірки дистанції та виконання затримки перезарядки ворог викликає атаку, яка завдає шкоду гравцю. У момент атаки активується анімація удару та надсилається сигнал для нанесення шкоди, якщо гравець перебуває в зоні ураження (ліст. 2.7).

Лістинг 2.7 – Атака ворога

```
private void AttackPlayer()
{
    if (targetPlayer == null || isDead) return;
    if (Time.time >= nextAttackTime)
    {
        animator.SetTrigger("AttackTrig");
        targetPlayer.GetComponent<Health>().TakeDamage(attackDamage);
        nextAttackTime = Time.time + attackCooldown;
    }
}
```

кінець лістингу 2.7

Отримання шкоди ворогом у грі реалізовано через метод TakeDamage(int amount). Коли ворог отримує удар, кількість його здоров’я (currentHP) зменшується на величину завданої шкоди. Якщо у ворога після цього ще залишається здоров’я, запускається анімація ушкодження через тригер HurtTrig. Якщо здоров’я знижується до нуля або нижче, викликається метод Die(), який запускає анімацію смерті та знищує об’єкт ворога через одну секунду (ліст. 2.8).

Лістинг 2.8 – Отримання шкоди ворогом

```
public void TakeDamage(int amount)
```

```
{
    if (isDead) return;
    currentHP -= amount;
    if (animator != null && currentHP > 0)
    {
        animator.SetTrigger("HurtTrig");
    }
    if (currentHP <= 0)
    {
        currentHP = 0;
        Die();
    }
}
```

кінець лістингу 2.8

Анімація смерті ворога реалізована у методі Die(), який запускається після зниження здоров’я ворога до нуля. При цьому встановлюється прапорець смерті, активується тригер анімації смерті (DeathTrig), після чого ворог видаляється зі сцени через одну секунду (ліст. 2.9).

Лістинг 2.9 – Анімація смерті ворога

```
private void Die()
{
    if (isDead) return;
    isDead = true;
    if (animator != null)
    {
        animator.SetTrigger("DeathTrig");
        Destroy(gameObject, 1f);
    }
    else
    {
        Destroy(gameObject);
    }
}
```

кінець лістингу 2.9

Отримання досвіду гравцем реалізовано методом GainExperience(int amount), який додає задану кількість досвіду до загального значення. Якщо кількість досвіду досягає або перевищує поріг для поточного рівня, автоматично викликається підвищення рівня (LevelUp()).Значення для порогу

рівня можна змінити всередині Unity за допомогою Inspector. Після кожної зміни досвіду генерується подія OnStatsChanged, яка дозволяє оновити інтерфейс чи виконати інші дії (ліст. 2.10).

Лістинг 2.10 – Отримання досвіду гравцем

```
public void GainExperience(int amount)
{
    experience += amount;
    if (experience >= GetRequiredExperience())
    {
        LevelUp();
    }
    OnStatsChanged?.Invoke();
}}
```

кінець лістингу 2.10

Таким чином, у процесі розробки гри «Легенда про Каїна» було проведено ґрунтовний аналіз вимог і побудовано чітку архітектуру програмного продукту. Реалізація ключових ігрових систем – від управління персонажем і бойової логіки до системи досвіду та обробки анімацій – базувалася на принципах модульності, незалежності компонентів та подальшої масштабованості. Такий підхід забезпечив стабільність роботи, зручність підтримки коду, а також відкрив широкі можливості для подальшого розширення ігрового функціоналу без критичних змін у структурі проєкту. Це дозволило максимально ефективно перейти від етапу проектування до повноцінної реалізації з мінімальними труднощами, заклавши надійну основу для розвитку гри в майбутньому.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

У процесі реалізації поставленої задачі було важливо вибрати оптимальні інструменти, підходи та алгоритми, які забезпечують ефективну розробку двовимірної гри з урахуванням актуальних вимог до продуктивності,

масштабованості та гнучкості розширення. Особливу увагу приділялося тому, щоб створений програмний продукт був не лише стабільним і зручним для користувача, а й технологічно сучасним, із потенціалом для розвитку.

В якості основного рушія для розробки гри обрано платформу Unity, яка на сьогодні є стандартом де-факто для інді-розробників та професійних команд. Серед ключових переваг Unity – гнучка система роботи зі сценами, підтримка Tilemap, багаторівнева інтеграція фізики, анімацій, UI-компонентів, потужна система Asset Store та розвинена документація. Використання мови програмування C# забезпечило впровадження об’єктно-орієнтованих практик та сучасних підходів до побудови ігрової логіки.

Для написання та редагування коду використовувалось середовище Visual Studio – одне з найзручніших IDE для розробки під Unity. Воно надає розширені можливості для автодоповнення, рефакторингу, виявлення помилок у реальному часі та ефективного налагодження, що значно прискорює розробку.

Під час реалізації гри використовувались такі методи:

– компонентно-орієнтований підхід: кожен об’єкт сцени складається з набору компонентів, які реалізують окремі аспекти поведінки або властивостей, що забезпечує простоту повторного використання та розширюваність логіки;

– делегування та події: для гнучкої взаємодії між незалежними модулями застосовуються делегати, callback-методи та події, що дозволяє мінімізувати жорсткі зв’язки між класами;

– об’єктно-орієнтоване програмування: ієрархія класів структурована так, щоб максимізувати читабельність коду та спростити спадкування;

– машини станів: поведінкові механіки для гравця та ворогів реалізовані через зміни стану (рух, атака, пошкодження, смерть тощо), що забезпечує зрозумілу логіку та легкість доопрацювань.

Для навігації ворогів у складному середовищі використовувався вбудований у Unity алгоритм NavMesh, який дозволяє динамічно розраховувати маршрути обходу перешкод. У разі, коли для певних об’єктів

застосування NavMesh було недоцільним, використовувалися власні прості алгоритми пересування з елементами випадкової орієнтації.

Бойова система базується на роботі з Collider2D і подіями тригерів: при зіткненні гравця з ворогом ініціюється механізм втрати здоров’я, а при досягненні нуля – запуск анімації смерті та відповідна обробка об’єкта. Система пошкоджень враховує затримку між атаками, короткочасне безсмертя після удару та наочний візуальний супровід (анімації, ефекти, зміна матеріалу).

Ігрове середовище формувалося за допомогою Tilemap, що дозволило швидко створювати рівні з різними шарами для води, землі, трави, перешкод і декоративних елементів. Колізії для непрохідних об’єктів налаштовувалися через TilemapCollider2D, що забезпечило коректну взаємодію персонажів із картою.

Графіка гри побудована на спрайтах та анімаціях, що контролюються Animator Controller. Для оптимізації продуктивності окремі вороги використовували спрощені матеріали замість стандартних аніматорів, що зменшувало навантаження на систему.

Користувацький інтерфейс створювався через систему Canvas та UI-компоненти Unity, із динамічним оновленням значень у реальному часі через відповідні скрипти. Це дозволило забезпечити інтуїтивну взаємодію гравця з грою та своєчасний візуальний зворотний зв’язок.

Завдяки такій структурі вибраних інструментів, підходів і алгоритмів, процес розробки гри був ефективним, технологічно сучасним і відкритим до майбутніх доопрацювань.

РОЗДІЛ 3
РОЗРОБКА ДВОВИМІРНОЇ ГРИ З ВИКОРИСТАННЯМ UNITY
1.3 Практична реалізація ігрових механік та інтерфейсу

У процесі розробки гри «Легенда про Каїна» було спроєктовано та реалізовано цілісну архітектуру 2D-гри (рис. 3.1), яка об’єднує всі основні компоненти ігрового процесу.

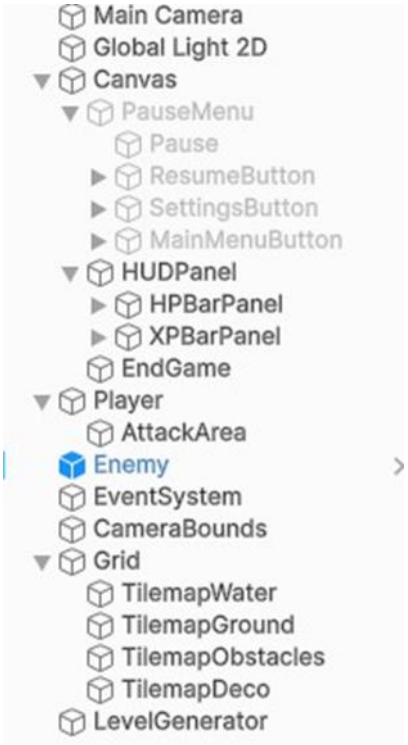

Рисунок 3.1 – Вигляд ієрархії гри

Створена структура охоплює ключові механіки дослідження ігрового світу, бойову систему, інтерактивність середовища та продуманий користувацький інтерфейс. Особлива увага приділялася гармонійному поєднанню логіки взаємодії об’єктів, гнучкості розширення функціоналу та зручності управління. Завдяки цьому гравець отримує змогу повноцінно досліджувати локації, взаємодіяти з NPC та предметами, брати участь у боях і контролювати персонажа у максимально інтуїтивному та зрозумілому середовищі.

Рух гравця реалізований у скрипті PlayerController через обробку введення з клавіатури. Кожен кадр гри відбувається зчитування вектору напрямку руху (метод GameInput.Instance.GetMovementVectorNormalized()), після чого значення передаються у фізичний компонент Rigidbody2D. Це забезпечує можливість вільного переміщення персонажа у чотирьох напрямках. Для плавності руху використовується метод MovePosition, який гарантує відсутність ривків і коректну взаємодію з колайдерами середовища.

Важливою частиною реалізації є відокремлення стану руху від стану спокою. Визначення, чи рухається гравець, здійснюється через перевірку модуля вектора введення: якщо значення більше нуля, змінна isRunning стає true. Це дозволяє правильно перемикати анімації залежно від дій гравця.

Система отримання шкоди у проєкті побудована на основі методу TakeDamage з компоненту Health. При отриманні удару від ворога значення currentHP зменшується на величину завданої шкоди. Якщо у персонажа залишається здоров’я, в аніматорі активується тригер ушкодження (HurtTrig). Якщо ж здоров’я зменшується до нуля, викликається метод Die, який запускає анімацію смерті (DeathTrig) і через 1 секунду видаляє об’єкт персонажа зі сцени. Ефект knockback у цій реалізації не використовується, натомість головний акцент зроблено на візуальних ефектах.

Анімації персонажа управляються компонентом Animator (рис. 3.2), до якого прив’язані три основні стани: спокій, рух та смерть.

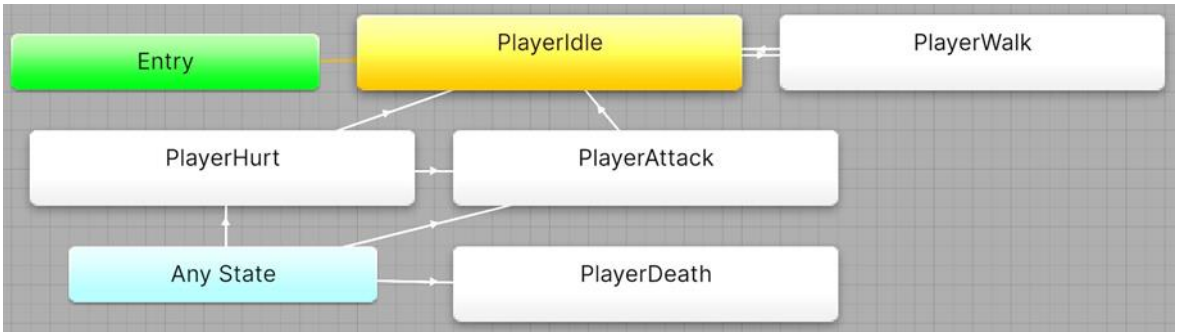

Рисунок 3.2 – Вигляд аніматора PlayerAnimation

Переходи між анімаціями здійснюються автоматично на основі булевої змінної isRunning (ходьба/стоп) та тригерів HurtTrig і DeathTrig (отримання ушкоджень і смерть). Завдяки цьому гравець завжди бачить актуальний стан персонажа: у разі руху – циклічна анімація ходьби, під час спокою – статична поза з ефектом дихання персонажа, а при смерті – фінальна анімація, що дозволяє юзеру краще занурюватись в ігровий процесс.

У ході розробки гри «Легенда про Каїна» особливий акцент було зроблено не лише на функціональність, а й на цілісне ігрове відчуття. Всі анімації персонажів синхронізовано з основною ігровою логікою, тож кожна дія має точний візуальний відгук. Зокрема, під час атаки гравця спочатку активується відповідна анімація через компонент Animator, і лише у визначений момент (найчастіше – у середині анімації) насправді відбувається нанесення шкоди ворогу. Це забезпечує максимально природнє та правдоподібне сприйняття бойових механік, а також мінімізує розбіжності між візуальними ефектами та ігровими подіями.

У проєкті впроваджена система подій і делегатів, яка дозволяє різним компонентам реагувати на ключові зміни стану без жорсткої прив’язки між об’єктами. Наприклад, у разі смерті гравця активується подія OnPlayerDeath, на яку підписані інші системи – це дозволяє автоматично відключити керування персонажем, запустити анімацію смерті, а також показати екран поразки. Такий підхід відповідає сучасним принципам слабкого зв’язування та підвищує масштабованість коду.

Орієнтація персонажа також враховує положення курсора миші або напрямок руху. Це забезпечує коректний розворот спрайта вліво чи вправо й дозволяє будувати інтуїтивну бойову систему, де атака завжди спрямована у відповідний бік.

Для ворогів, наприклад, гобліна, розроблено окремий набір станів і відповідних анімацій у Animator: патрулювання, переслідування гравця, атака та смерть, для інших ворогів система аналогічна, що спрощує додавання будь яких інших, також дозволяє активно використовувати Prefab. Перехід між

цими станами керується логікою штучного інтелекту ворога (EnemyAI), що робить кожного противника більш «живим» та візуально виразним (рис. 3.3).

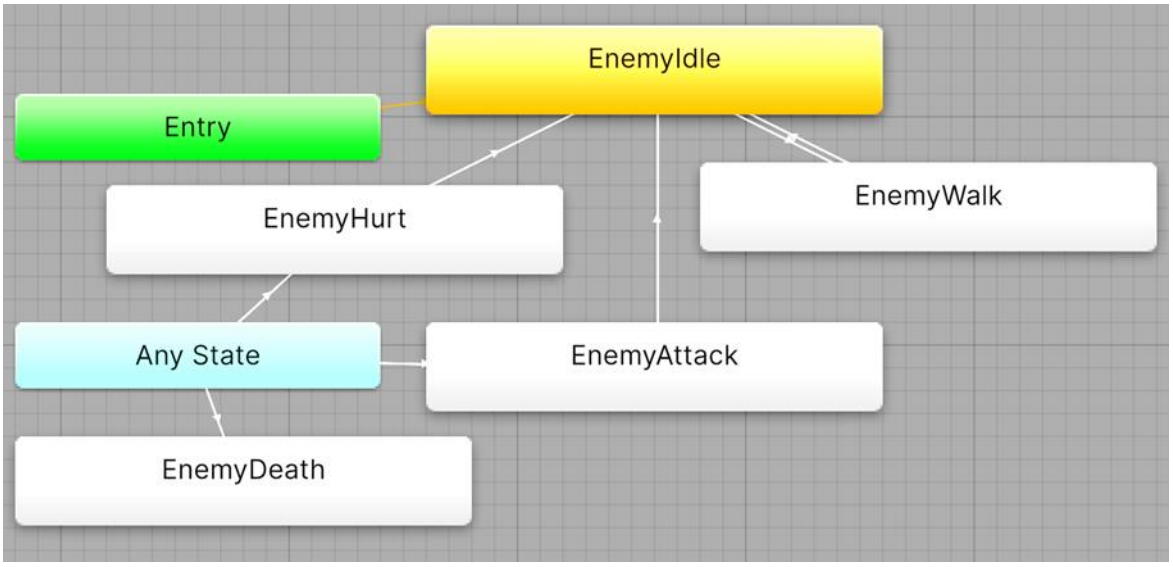

Рисунок 3.3 – Вигляд аніматора GoblinAnimation

Такий підхід до організації анімацій, логіки подій і станів забезпечив гнучкість і масштабованість проекту, а також підвищив якість геймплейного та візуального досвіду.

Для ворога-гобліна у моєму проєкті використано класичний підхід до анімації за допомогою компонента Animator. Логіка гобліна структурована подібно до гравця: у скрипті ворога при зміні станів (рух, атака, отримання шкоди, смерть) викликаються відповідні тригери або булеві параметри аніматора. Завдяки цьому для гобліна реалізовано плавну зміну анімацій – руху, атаки, ушкодження й смерті – що робить його поведінку візуально зрозумілою й живою.

Перемикання між анімаціями здійснюється автоматично залежно від логіки поведінки ворога у скриптах. Наприклад, при переході в стан атаки у скрипті викликається тригер атаки (SetTrigger("AttackTrig")), а при отриманні шкоди – тригер ушкодження (SetTrigger("HurtTrig")). Аналогічно, при зниженні здоров'я до нуля запускається анімація смерті (SetTrigger("DeathTrig")), після завершення якої об'єкт ворога

видаляється зі сцени.

Такий підхід дозволяє досягти як високої продуктивності, так і якісного візуального супроводу. Гоблін виглядає органічно у геймплеї, а його анімації коректно синхронізуються з діями та станами, що підвищує занурення у гру та робить бої динамічними.

Для проєкті для організації динамічного слідкування камери за головним героєм було реалізовано власний скрипт FollowTarget. На відміну від використання готових рішень на кшталт Cinemachine, обрано простий і зрозумілий підхід, який повністю задовольняє потреби гри та дозволяє досягти плавного ефекту стеження.

Скрипт FollowTarget прив’язується до камери та отримує посилання на об’єкт гравця. Щоразу при оновленні кадру (у методі LateUpdate) камера обчислює бажану позицію, використовуючи координати гравця і визначене зміщення (offset). Для плавності руху між поточною і цільовою позицією використовується інтерполяція, наприклад, через функцію Vector3.Lerp або схожий алгоритм. Завдяки цьому переміщення камери виглядає природно, без різких стрибків чи смикань, а гравець завжди залишається у фокусі незалежно від швидкості свого руху.

Такий підхід вирізняється насамперед мінімалістичністю – код залишається максимально простим для сприйняття, легко піддається редагуванню й не обтяжений сторонніми залежностями чи зайвими пакетами. Окрім цього, підвищується гнучкість, оскільки всі ключові параметри, такі як швидкість згладжування або величина зміщення, доступні для оперативної зміни безпосередньо через інспектор Unity, що значно пришвидшує ітерації під час налаштування. Ще однією важливою рисою стає оптимізація – оскільки відсутній зайвий функціонал, система працює максимально ефективно та не створює конфліктів із іншими компонентами проекту, дозволяючи зберегти високу продуктивність навіть у складних ігрових сценах.

Власна система стеження за гравцем відмінно інтегрується з іншими елементами проєкту, зокрема UI та логікою зміни сцен. За потреби легко

розширити функціонал, наприклад, обмежити межі руху камери, додати ефекти масштабу чи інші додаткові можливості.

Такий підхід повністю задовольнив вимоги проєкту: забезпечив комфортний і стабільний огляд для гравця, зберіг чистоту архітектури й дозволив гнучко налаштовувати поведінку камери відповідно до сценаріїв гри.

У моєму проєкті система ворогів побудована на основі чітко визначених станів, які дозволяють організувати просту, але ефективну поведінку противників. Кожен ворог у грі може перебувати у кількох базових станах: очікування, патрулювання, переслідування гравця або атака.

Перехід між цими станами відбувається автоматично, залежно від ситуації на ігровій сцені – зокрема, від відстані до головного героя. Наприклад, коли гравець наближається до ворога на визначену відстань, ворог переходить із режиму патрулювання у стан переслідування. Якщо ж герой ще більше наближається, ворог активує атаку.

Механіка атаки ворога враховує затримку між ударами: після кожної атаки встановлюється таймер відновлення, який не дозволяє наносити шкоду занадто часто. Це дозволяє збалансувати бої та уникнути ситуацій, коли ворог міг би атакувати кілька разів поспіль без паузи.

Завдяки такій системі поведінки вороги у грі здаються «живими»: вони реагують на дії гравця, можуть змінювати свою поведінку залежно від ситуації та створюють відчуття динамічної взаємодії у процесі гри.

Система переходів між сценами у грі реалізована таким чином, щоб створити ілюзію цілісного ігрового світу. Кожна локація представлена як окрема сцена Unity, яка завантажується при досягненні гравцем спеціальної зони переходу (наприклад, входу у двері, портал чи межу карти). Для цього використовується спеціальний об’єкт-тригер, що реагує на зіткнення з гравцем та ініціює завантаження нової сцени через SceneManager.

Щоб зробити перехід між сценами плавним і уникнути раптових візуальних змін або затримок, під час зміни локації застосовується ефект затемнення екрана (fade-in/fade-out). Після повного затемнення сцена

перемикається, і гравець опиняється у новій точці входу, визначеній для цієї локації. Такий підхід дозволяє створити ефект відкритого світу без видимих пауз у геймплеї та підвищує якість сприйняття переходів.

Інтерфейс користувача (UI) у грі розроблено з урахуванням адаптивності. Для цього всі елементи розміщені на Canvas із гнучкими налаштуваннями масштабування та анкерування. Canvas автоматично підлаштовується під різні роздільності та співвідношення сторін екрана, що забезпечує коректне відображення інтерфейсу як на великих моніторах, так і на невеликих ноутбуках. Завдяки цьому UI-елементи не перекриваються, не виходять за межі екрану та залишаються зручними для користувача навіть при зміні розміру вікна гри.

У моєму проєкті переміщення ворогів реалізовано за допомогою простого алгоритму переслідування цілі. Замість використання складних навігаційних систем, таких як NavMesh, логіка руху побудована на постійному оновленні позиції ворога відносно позиції гравця.

У скрипті ворога кожного кадру обчислюється напрямок руху до гравця (direction = (player.position – transform.position).normalized) і позиція ворога змінюється на задану величину із врахуванням швидкості. Для цього може використовуватись метод Vector2.MoveTowards, пряме додавання зміщення до transform.position або рух через Rigidbody2D. Такий підхід є ефективним для 2D-ігор і дозволяє отримати просту, передбачувану й керовану поведінку.

Перевірка відстані до гравця визначає, у якому стані перебуває ворог: якщо гравець далеко, ворог патрулює територію або перебуває у режимі очікування; якщо наближається – починає переслідування; а якщо знаходиться поруч, переходить у стан атаки. Завдяки цій системі противники динамічно реагують на дії гравця, що робить гру більш захопливою.

Такий підхід легко масштабується – для кожного типу ворога можна задати унікальну швидкість, радіус виявлення або навіть власний алгоритм руху. У результаті вороги поводяться природно, але код залишається простим

і зрозумілим для подальшого розширення.

Загалом, розроблені ігрові механіки свідчать про гнучкість і ефективність вибраних технологій – рушія Unity та мови програмування C#. Усі системи гри були побудовані з урахуванням принципів модульності: основна логіка, керування персонажем, анімації, бойова система, UI та механіка переміщення розділені на окремі компоненти.

Така архітектура значно спрощує підтримку й подальший розвиток проєкту. При потребі можна додавати нові можливості, ворогів, предмети, локації чи унікальні механіки, не вносячи критичних змін до вже реалізованого коду. Компонентна структура також дозволяє швидко оновлювати або замінювати частини гри без ризику порушити її цілісність.

Подібний підхід забезпечує надійність, стабільність та масштабованість проекту, що є особливо важливим для ігор із потенціалом подальшого розширення чи підтримки. Це відкриває широкі можливості для розвитку гри, внесення змін за відгуками гравців, а також для навчальних і практичних експериментів у майбутньому.

3.2 Тестування та виправлення помилок

Завершальним етапом створення гри «Легенда про каїна» стало комплексне тестування, що є невід’ємною частиною будь-якої якісної розробки. Цей процес дозволив не лише оцінити загальну працездатність реалізованих механік, а й віднайти приховані помилки, недоопрацювання у логіці взаємодії об’єктів, а також випробувати зручність користувача. Я перевіряв абсолютно все – від руху гравця і атак ворогів до коректності переходу між сценами та поведінки інтерфейсу на різних екранах.

Тестування гри відбувалося інтерактивно, передусім у середовищі Unity, де можна було відразу бачити ефект кожної зміни, але й обов’язково у фінальній збірці під Windows, адже багато багів проявляються лише при запуску проєкту поза редактором. Особливу увагу я приділив ручному

тестуванню – не тільки через обмеженість команди, а й тому, що такий підхід дозволяє максимально швидко перевірити роботу кожного геймплейного модуля на практиці.

Під час комплексного тестування «Легенда про Каїна» довелося зіткнутися з низкою характерних для Unity-розробки проблем, більшість із яких проявлялися не одразу, а лише у процесі багаторазових проходжень та імітації різних ігрових сценаріїв. Зі зростанням складності проєкту «Легенда про Каїна» тестування поступово відкривало нові пласти неочевидних, але дуже впливових проблем. На перший погляд усе працювало справно, однак під час справжнього геймплею проявлялися баги, які неможливо було виявити лише статичним аналізом коду. Під час швидких переміщень гравця між локаціями, наприклад, виникала ситуація, коли він міг випадково активувати одразу кілька тригерів переходу – або ж взагалі потрапити в так звану «сіру зону» між сценами. Це спричиняло повторне завантаження тієї ж самої локації, появу дубльованих персонажів або зникнення інтерфейсу. Розв’язанням стала реалізація затримки активації тригера, а також короткочасне блокування керування гравцем під час зміни сцен – ці заходи гарантували унеможливлення багатократних переходів та забезпечили коректність роботи всієї системи переміщень.

Немало проблем підкинуло й одночасне ведення бою з кількома ворогами. Якщо два або більше супротивників атакували гравця разом, іноді спрацьовувала багатократна анімація отримання шкоди, через що персонаж починав «блимати», а система невразливості не встигала коректно блокувати додаткову шкоду. Це призводило до миттєвої втрати здоров’я й відчуття несправедливості в бою. Щоб виправити ситуацію, була повністю переглянута система флагів, таймерів і взаємодії зі станами – тепер шкода проходить лише після завершення поточної анімації та гарантованої паузи невразливості. Це зробило бої більш чесними та контрольованими для гравця.

Зустрічались і проблеми з тайлмепами – після додавання нових декорацій або змін середовища виявлялось, що гравець може провалитись у

край сцени чи застрягти у тайлі, який мав бути прохідним. Джерело помилки виявилось у невірно налаштованих Collider Type і шарів. Після кількох ревізій та перевірки кожного тайлу в Tile Palette ця проблема була усунута й механіка руху стала стабільною.

Нерідко складнощі виникали й зі звуком: дублювання ефектів, затримки відтворення або їх несподіване зникнення ставали відчутним недоліком під час інтенсивних битв. Аналіз коду показав, що виклики аудіо були розкидані між анімаційними подіями й окремими скриптами, що породжувало хаос. Було прийнято рішення централізувати всі звукові події через AudioManager, завдяки чому вдалося синхронізувати ефекти з діями та позбутися багатократних програвань.

Під час редагування системи управління виявилося, що після зміни розкладки клавіш або оновлення InputManager гравець міг тимчасово втратити контроль над персонажем. Вирішити цю проблему вдалося через явне перевантаження системи введення після повернення до гри з меню налаштувань, а також через додаткові перевірки на старті кожної сцени. Це значно підвищило зручність і надійність управління.

Складніше було помітити баг із повторним нарахуванням досвіду. Якщо гравець швидко змінював сцени, XP міг зараховуватись декілька разів поспіль через дубльовані підписки на події знищення ворога. Для вирішення проблеми було впроваджено централізований менеджер досвіду та чітку систему відписки, яка унеможливила виникнення «фантомних» нарахувань XP.

Зіткнувшись із бажанням адаптувати гру до мобільних платформ, вдалося знайти ще кілька неочевидних вад. Інтерфейс, розроблений під десктоп, погано масштабувався на сенсорних екранах, а багато елементів управління залишалися непридатними для торкання. Це підкреслило важливість гнучких систем введення та адаптивного UI вже на етапі архітектури проекту.

Траплялися й менш масштабні, але не менш впливові баги. На ранніх етапах проєкту виявилось, що персонаж міг рухатися під час смерті або поки

триває анімація отримання шкоди. Причина була у неправильному скиданні прапора руху, тож після оновлення логіки станів усі дії стали синхронізованими з анімаціями. Інший цікавий випадок стосувався бойової системи: при надто швидкому натисканні кнопки атаки хітбокс не завжди встигав активуватись у потрібний момент, і ворог не отримував шкоди, хоча анімація відпрацьовувала. Додання Animation Events та точна прив’язка моменту удару до кадру анімації вирішили це питання остаточно.

Проблеми проявлялися й під час переходів між сценами: вороги не завжди коректно знищувалися, і інколи при поверненні до раніше відвіданої локації ворогів могло стати удвічі більше. Впровадження чіткого контролю життєвого циклу об’єктів і відмова від зайвих DontDestroyOnLoad допомогли зберегти стабільність ігрового світу.

Не оминув увагу і інтерфейс: на широких або нестандартних екранах індикатори здоров’я, досвіду чи елементи HUD могли накладатися один на одного, зникати або розтягуватись. Глибше налаштування Canvas Scaler, а також впровадження Layout Group допомогли вирівняти й адаптувати інтерфейс під будь-яку роздільність.

Окремо варто згадати баги в бойовій системі ворогів. Іноді гоблін атакував гравця у зворотному напрямку або взагалі – не дивлячись на ціль, якщо координати майже збігались. Для виправлення введено додаткову логіку визначення напрямку атаки з урахуванням мінімальної різниці координат, що дозволило ворогам поводитись значно природніше.

В процесі розробки й тестування інколи виникали нестандартні ситуації, коли кілька помилок поєднувались, а наслідки проявлялися неочікувано: наприклад, досвід не оновлювався після знищення ворога, якщо дія відбувалась одразу після переходу сцени, або гравець міг отримати ушкодження від декількох ворогів одночасно й миттєво загинути. Постійний аналіз таких ситуацій, впровадження захистів, централізованих менеджерів та додаткових перевірок не лише виправили знайдені недоліки, а й зробили архітектуру гри більш надійною та підготували її до майбутнього

масштабування. Завдяки цьому досвіду структура коду стала чистішою, а ігровий процес – стабільнішим і зрозумілішим для гравця.

3.3 Структура ігрових ассетів та програмного коду

Графічне оформлення 2D-ігор ґрунтується насамперед на використанні спрайтів – двовимірних зображень, які виступають візуальним представленням усіх ігрових об’єктів на екрані. У моєму проєкті спрайти стали не просто елементом декору, а справжньою основою для створення цілісного віртуального світу. Саме за допомогою ретельно підібраних спрайтів було змодельовано як основних персонажів, так і їхніх супротивників, елементи інтерфейсу користувача. Вдале використання спрайтів не лише визначає художній стиль проєкту, підкреслює його унікальність, а й безпосередньо впливає на функціональну логіку гри, оскільки через візуальні зміни гравець отримує сигнал про стан ігрових об’єктів, їхню взаємодію чи небезпеку.

Усі елементи гри, які мають графічне утілення, реалізовані через один або декілька спрайтів, і вони поділяються за призначенням і способом використання. Найбільш простими й водночас найбільш поширеними є статичні спрайти. Вони не змінюють свого вигляду протягом гри й використовуються насамперед для фонових об’єктів: це різноманітні ландшафти, будівлі, дерева, кущі та інша рослинність, що не взаємодіє безпосередньо з гравцем, але створює потрібний настрій і підкреслює атмосферу обраного жанру.

Другою, ще більш помітною категорією є анімовані спрайти. Саме вони відповідають за динаміку й живість ігрового процесу. Анімовані спрайти застосовуються для персонажа гравця, ворогів, NPC, а також для різних об’єктів, що змінюють свій стан під час гри – наприклад, для порталів, скринь, пасток чи особливих ефектів, які реагують на дії користувача. Реалізація анімації в Unity відбувається за допомогою потужної системи Animator та

Animation Clip: набір окремих зображень (кадрів) поєднується у циклічні або тригерні анімації, які автоматично перемикаються в залежності від стану чи поведінки об’єкта. Такий підхід не лише забезпечує плавний та виразний рух, а й дозволяє швидко додавати нові стани або розширювати анімаційний спектр у разі масштабування гри. Це робить візуальне наповнення проєкту не просто привабливим, а й гнучким у розробці й підтримці.

Для головного героя у моєму проєкті створено п’ять основних анімаційних станів:

– PlayerIdle – спокій;
– PlayerWalk – рух;
– PlayerHurt – отримання шкоди;
– PlayerDeath – смерть;
– PlayerAttack – атака (рис. 3.4).

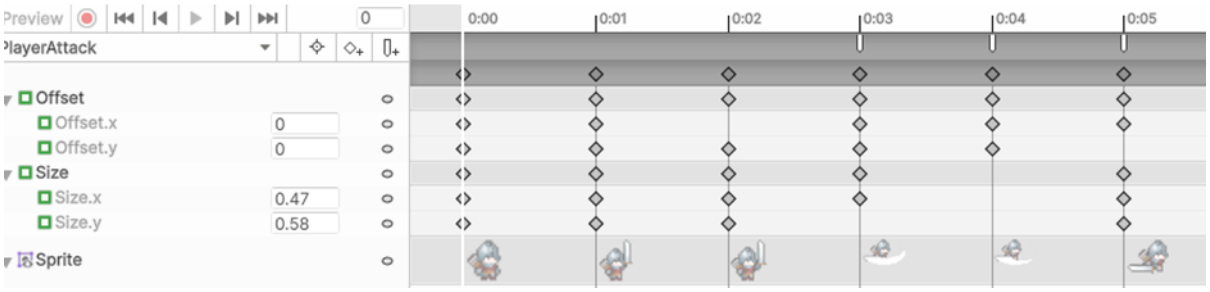

Рисунок 3.4 – Налаштування анімації для PlayerAttack

Перемикання між цими анімаціями відбувається автоматично на основі внутрішньої логіки гри та змінних у Animator Controller. Наприклад, при отриманні шкоди активується PlayerHurt, при початку руху – PlayerWalk, а при смерті – PlayerDeath.

У моїй грі для всіх типів персонажів – як для гравця, так і для ворогів – використовується єдина система анімації на базі Animator. Це рішення дозволило створити чітку, логічно структуровану й легко підтримувану архітектуру, де кожен персонаж керується набором основних анімаційних станів. Зокрема, для кожного з них налаштовано такі базові анімації: спокій

(Idle), рух (Walk), атака (Attack), отримання ушкодження (Hurt) і смерть (Death).

Завдяки такому підходу, додавання нових ворогів або NPC не потребує створення унікальних анімаційних контролерів чи складної додаткової логіки. Достатньо під'єднати до об'єкта існуючий контролер Animator та призначити відповідні спрайти для потрібних станів. Весь процес роботи з анімаціями відбувається однаково для будь-якого персонажа: під час руху активується стан Walk, під час атаки чи отримання шкоди – відповідні тригери або параметри, а при смерті вмикається фінальна анімація Death. Така уніфікація не лише оптимізує робочий процес під час розробки, а й забезпечує цілісність художнього стилю гри.

На рівні реалізації до кожного персонажа додається компонент SpriteRenderer для рендерингу спрайтів, Animator для керування станами, а також Rigidbody2D і Collider для забезпечення коректної взаємодії з фізичним середовищем. Усі анімації створені з послідовностей спрайтів, а переходи між ними налаштовуються у Animator Controller через зміни булевих параметрів або тригерів, які викликаються відповідними скриптами залежно від ігрової логіки.

Завдяки цьому підходу, весь процес керування анімацією став максимально прозорим і універсальним: зміни або оновлення станів одразу застосовуються до всіх ігрових персонажів, а додавання нових ворогів або ігрових ситуацій не потребує радикальних змін у коді чи структурі проєкту.

Чітко продумана й структурована організація проєкту стала основою ефективної розробки гри, її тестування, підтримки та подальшого масштабування. Ще на етапі планування була визначена необхідність суворо дотримуватися принципів модульності – як у структурі ресурсів, так і в організації програмного коду. Це дозволило розділити всі елементи гри за логічними ознаками й зберігати їх у відповідних директоріях.

Наприклад, у папці Animations розміщені всі анімаційні кліпи та контролери, що керують поведінкою гравця, ворогів і різноманітних ігрових

об’єктів. Усі префаби – наперед налаштовані шаблони персонажів, ворогів, предметів, дверей, точок переходу між локаціями – зберігаються у папці Prefabs

Візуальні ресурси – спрайти, текстури, палітри тайлів – акуратно згруповані у папці Sprites, що дозволяє швидко знаходити й змінювати художні елементи при необхідності. Сценарії, які реалізують ігрову логіку, винесені у Scripts й додатково розділені за категоріями (наприклад, окремо для гравця, ворогів, UI та менеджерів сцен). По такій ж аналогії існує папка Animations, в якій знаходяться анімації гравця та ворогів.

Завдяки такій структурі проєкту кожен елемент легко знайти, перевикористати чи адаптувати під нові потреби гри (рис. 3.5).

Рисунок 3.5 – Повна структура папок Assets

Це особливо важливо при подальшому розвитку – можна швидко інтегрувати новий функціонал, додати ворогів або ігрові механіки без ризику

порушити цілісність чи заплутати структуру. Організація папок і чітке розмежування ролей між файлами стали запорукою гнучкості й керованості усього проєкту. Важливою складовою якісної розробки гри є чітка і продумана організація проєкту на всіх рівнях – від структури ассетів до структури програмної логіки.

У роботі над грою я з самого початку дотримувався принципу зрозумілого й прозорого структурування, що дозволяє не лише ефективно керувати ресурсами під час створення гри, а й забезпечує простоту подальшої підтримки й масштабування проєкту.

Всі ігрові ресурси розподілені по тематичних папках відповідно до їхнього призначення. Так, анімаційні кліпи та контролери містяться в окремій директорії, що значно полегшує доступ до будь-якої анімації гравця або ворога, коли потрібно внести зміни, замінити фрагмент або підключити додаткову послідовність. Завдяки цьому можна швидко впроваджувати нові ідеї чи покращення без ризику зламати вже наявний функціонал.

Готові ігрові об’єкти, які використовуються неодноразово, створюються у вигляді префабів. Це дозволяє, наприклад, розмістити однакових ворогів на різних рівнях, або забезпечити консистентність для всіх переходів між сценами. В будь-який момент можливо відредагувати шаблон – і зміни автоматично застосуються до всіх його копій у грі, що суттєво прискорює процес доопрацювання.

Весь програмний код зберігається в окремій папці Scripts. Усередині вона логічно поділена на підрозділи за тематикою: окремо розташовані скрипти для керування гравцем, ще в іншій частині – скрипти ворогів, далі йде логіка інтерфейсу та глобальних менеджерів. Такий поділ забезпечує ізольованість функціоналу, мінімізує ризик неочікуваних залежностей та дозволяє працювати над різними системами паралельно. Наприклад, зміна механіки атаки ворога жодним чином не вплине на логіку меню чи налаштувань, і навпаки.

Графічні елементи проєкту представлені в папці Sprites – тут зібрані всі

спрайти для персонажів, ворогів, предметів, а також деталі ігрового оточення. Для реалізації карт і локацій використовується Tilemap – вся відповідна інформація, включаючи тайли, палітри та мапи, зберігається у відповідній папці, що робить процес створення і редагування рівнів швидким і зручним. Додатково виділені директорії для шрифтів, графіки, матеріалів, налаштувань та інших допоміжних ресурсів. Окрема увага приділена правильній організації сцен – кожна ігрова локація, головне меню, проміжні екрани зберігаються у Scenes. Завдяки такій структурі легко зрозуміти, які сцени використовуються у грі й як швидко їх знайти для редагування.

Особливістю побудови програмної логіки стало використання принципів об’єктно-орієнтованого програмування. Кожен клас або компонент чітко відповідає за свій шматок функціоналу – наприклад, EnemyAI керує поведінкою ворогів, GameManager бере на себе глобальне управління станом гри, переходами між сценами та ініціалізацією основних систем. Впровадження патерну Singleton для основних менеджерів гарантує, що доступ до цих об’єктів завжди унікальний і централізований, а значить – відсутній дублюючий або зайвий код.

Важливим етапом стало впровадження системи делегатів і подій. Завдяки цьому різні частини гри взаємодіють між собою без жорсткої зв’язаності: наприклад, смерть гравця не безпосередньо керує інтерфейсом чи ворогами, а лише генерує відповідну подію, на яку підписані всі зацікавлені об’єкти. Це забезпечує як чистоту архітектури, так і простоту внесення змін у майбутньому. З технічної точки зору кодова база була організована з урахуванням принципів розділення відповідальностей і повторного використання. Базова механіка (рух, атака, отримання шкоди) винесена у спільні компоненти або базові класи, від яких спадкуються гравець, вороги й інші ігрові об’єкти. Подібний підхід дозволяє додавати новий функціонал, не переписуючи основну логіку, а лише розширюючи її у спеціалізованих класах.

Велику роль у підтримці якості та керованості коду відіграє також інтеграція з Visual Studio. Це середовище надає розширені можливості для

дебагу, автоматичного завершення коду, швидкого переходу до визначень класів і методів, що суттєво прискорює пошук і виправлення помилок, а також робить процес розробки більш ефективним.

Структурованість ассетів, чіткий поділ коду та логічна організація всіх компонентів дозволили уникнути хаосу й плутанини навіть на пізніх стадіях розробки. Завдяки цьому будь-яка модифікація чи розширення – наприклад, додавання нових ворогів або впровадження нової ігрової механіки – відбуваються ізольовано й не загрожують стабільності основного проєкту. Такий підхід є одним із визначальних чинників успіху проєкту й гарантує, що гра буде підтримуватися й розвиватися без надмірних витрат часу чи ресурсів у майбутньому.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи була проведена ретельна аналітична ревізія сучасних підходів до створення двовимірних ігор, із фокусом на механіки та особливості жанру RPG. Я уважно проаналізував різноманітні архітектурні патерни, які лежать в основі побудови сцен і внутрішньої структури світу, а також дослідив логіку організації взаємодії між головним героєм, неігровими персонажами та об’єктами оточення. Був зроблений акцент на багаторівневій організації ігрового простору: від глобальної побудови сцен до нюансів розташування колайдерів, спавнів ворогів і розміщення тригерів для інтерактивних подій. Особливе місце в дослідженні посіла сучасна система анімації, що базується на принципах переходів між станами та інтеграції анімаційного контролера з ігровою логікою. Було проаналізовано не лише типовий цикл життя гравця, але й алгоритми штучного інтелекту ворогів, які забезпечують як мінімальне патрулювання та переслідування, так і складніші типи поведінки з реакцією на різні дії користувача.

Паралельно з аналізом існуючих рішень був сформований детальний перелік як функціональних, так і нефункціональних вимог до майбутньої гри. Питання продуктивності та стабільності роботи проекту стали першочерговими – особливо з урахуванням можливого масштабування функціоналу в майбутньому. Особливий акцент було зроблено на адаптивність інтерфейсу під різні пристрої та роздільності, зручність взаємодії для гравця та гнучкість самої архітектури. Я детально пропрацював концепції розширюваності, так щоб будь- які майбутні доповнення – нові вороги, квести чи функції – легко вписувалися у вже існуючий каркас гри без потреби переписувати значні обсяги коду.

Вибір технологій розробки був не менш зваженим: рушій Unity надав усі необхідні інструменти для створення складних 2D-локацій, опрацювання фізики, анімаційних процесів, взаємодії з UI та організації роботи з тайловими

картами. Мова C# обрана не лише через повну інтеграцію з Unity, а й завдяки її сучасному синтаксису, який ідеально підходить для побудови об’єктно-орієнтованих структур та дозволяє підтримувати чисту і зрозумілу архітектуру коду. Середовище Visual Studio забезпечило зручну роботу з проектом, включаючи можливість налагодження в реальному часі, підсвітку синтаксису, швидку навігацію по класах та автоматичне завершення коду, що пришвидшило розробку та знизило кількість помилок на етапі написання.

В ході розробки було повністю реалізовано механіку керування головним персонажем: від базового пересування й плавної взаємодії з фізичними об’єктами до складніших сценаріїв реакції на атаки ворогів, активації тригерів та вирішення колізій. Велика увага приділялася дрібницям – таким, як природність переміщення, коректне відтворення анімацій під час зміни станів чи точність взаємодії зі складними елементами середовища.

Розробка бойової системи включала створення ефективної логіки нанесення й отримання шкоди та побудову коректної анімаційної реакції на удари. Було продумано сценарії смерті персонажів, обробку ситуацій, коли гравець може бути атакований кількома ворогами одночасно, і синхронізацію цих подій із візуальним та звуковим фідбеком.

Штучний інтелект ворогів реалізований як адаптивна система з кількома рівнями поведінки. Вороги здатні не лише патрулювати територію й переслідувати гравця, а й змінювати стани залежно від дій гравця, відстані та поточного сценарію. Досягнуто динамічного балансу між викликом для гравця та справедливістю бою, а багаторівнева система переходів між станами дозволила зробити кожен тип супротивника унікальним.

Окремо опрацьовано побудову локацій: використано кілька типів тайлів для створення основної поверхні, перешкод, водойм і декоративних елементів. Реалізовано гнучке керування межами рівня, що унеможливлює вихід гравця за межі карти й дозволяє враховувати особливості кожної сцени. Для побудови складних сцен було залучено систему тайлових палітр, що дозволила експериментувати з різними варіантами оформлення ландшафту.

Не менш важливим завданням стало налаштування системи сцен і менеджменту ігрового процесу: зручно реалізовано початкове меню, екран паузи, плавний перехід між рівнями, автоматичне збереження й передачу необхідних даних. Забезпечено цілісність ігрового досвіду завдяки відсутності затримок та візуальних стрибків при переходах.

Завершальним етапом стало комплексне тестування усіх реалізованих механік. Кожен компонент – від анімаційного контролера до інтерфейсної логіки – пройшов багаторівневу перевірку на стабільність, продуктивність та коректність взаємодії з іншими елементами системи. Складні сценарії, що виникали під час гри, були проаналізовані й виправлені, завдяки чому проект досягнув високого рівня якості та підготовленості до подальшого масштабування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ember Arcade. Game Art & Assets. URL: https://www.emberarcade.com/home (дата звернення: 27.04.2025).

2. GameDev StackExchange. Unity 2D Questions. URL: https://gamedev.stackexchange.com/questions/tagged/unity2d (дата звернення:10.05.2025).

3. GitHub. Unity 2D Projects. URL: https://github.com/search?q=unity+2d+game&type=repositories (дата звернення: 22.02.2025).

4. Kenney.nl. Free Game Assets. URL: https://kenney.nl/assets (дата звернення: 04.02.2025).

5. Microsoft Learn. Programming in C#. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 25.03.2025).

6. Microsoft Visual Studio. URL: https://visualstudio.microsoft.com (дата звернення: 15.03.2025).

7. Pixelfrog. Tiny Swords Asset Pack. URL: https://pixelfrog-assets.itch.io/tiny-swords (дата звернення: 30.03.2025).

8. Red Blob Games. Game development algorithms and mechanics. URL: https://www.redblobgames.com/ (дата звернення: 19.04.2025).

9. Unity Asset Store. 2D Art & Animation. URL: https://assetstore.unity.com/2d (дата звернення: 14.02.2025).

10. Unity Hub. Download Unity. URL: https://unity.com/download (дата звернення: 02.05.2025).

11. Unity Learn. Create with Code (2D Pathway). URL: https://learn.unity.com/pathway/create-with-code (дата звернення: 28.04.2025).

12. Unity Technologies. C# API documentation. URL: https://docs.unity3d.com/ScriptReference/ (дата звернення: 12.02.2025).

13. Unity Technologies. Tilemap and Grid. URL: https://docs.unity3d.com/Manual/Tilemap.html (дата звернення: 07.05.2025).

14. Unity Technologies. Unity Manual. 2D Game Creation. URL: https://docs.unity3d.com/Manual/2D.html (дата звернення: 09.04.2025).

15. Zerie. Tiny RPG Character Asset Pack. URL: https://zerie.itch.io/tiny-rpg-character-asset-pack (дата звернення: 16.03.2025).